AUTOMATIC MODELING AND SIMULATION OF NETWORKED COMPONENTS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Nathaniel William Bruce

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

May 2011

Purdue University

Indianapolis, Indiana

APPENDIX

APPENDIX
SOURCE CODE

This appendix contains source code for the implementation of the methods in this thesis. All code is written in C++ using Visual Studio 2008.

A.1 Core Code

A.1.1 main.cpp

```
// main.cpp : Defines the entry point for the console application.
//

#include "stdafx.h"
#include "frmSignalSorting.h"
#include "FiniteStateMachine.h"
#include "FsmSimulation.h"
#include "frmSimulate.h"

void display_progress(float p, char* s);

[STAThreadAttribute]
int _tmain(int argc, _TCHAR* argv[])
{

    // load the dbc file
    Windows::Forms::OpenFileDialog^ ofd = gcnew Windows::Forms::OpenFileDialog();
    ofd->Title = "Open DBC File";
    ofd->FileName = "";
    ofd->Filter = "DBC Files (*.dbc)|*.dbc|All Files (*.*)|*.*";
    if (ofd->ShowDialog() != Windows::Forms::DialogResult::OK) return 0;
    char* dbc_path = (char*)(void*)Runtime::InteropServices::Marshal
        ::StringToHGlobalAnsi(ofd->FileName);
    can_dbc.ReadDbcFile(dbc_path);
    Runtime::InteropServices::Marshal::FreeHGlobal(IntPtr(dbc_path));
```

```
if ( (argc > 1) && (0 == wcscmp(argv[1], L"-synthesize")) )
{
    // allow user to sort the signals
    thesis_can::frmSignalSorting^ ss = gcnew thesis_can::frmSignalSorting();
    if (ss->ShowDialog() != Windows::Forms::DialogResult::OK) return 0;

    // process the clg trace file
    ofd->Title = "Open Communications Trace";
    ofd->FileName = "";
    ofd->Filter = "CLG Files (*.clg)|*.clg|All Files (*.*)|*.*";
    if (ofd->ShowDialog() != Windows::Forms::DialogResult::OK) return 0;
    char* clg_path = (char*)(void*)Runtime::InteropServices::Marshal
        ::StringToHGlobalAnsi(ofd->FileName);

    FiniteStateMachine fsm2;
    fsm2.Synthesize(clg_path, &can_dbc, display_progress);

    Runtime::InteropServices::Marshal::FreeHGlobal(IntPtr(clg_path));

    Windows::Forms::SaveFileDialog^ sfd = gcnew Windows::Forms::SaveFileDialog();
    sfd->Title = "Save Image File";
    sfd->FileName = "";
    sfd->Filter = "PNG Image (*.png)|*.png|Graphics Interchange Format (*.gif)|*.gif"
        + "|Bitmap Image (*.bmp)|*.bmp|All Files (*.*)|*.*";
    if (sfd->ShowDialog() == Windows::Forms::DialogResult::OK)
    {
        char* img_path = (char*)(void*)Runtime::InteropServices::Marshal
            ::StringToHGlobalAnsi(sfd->FileName);
        if (sfd->FilterIndex == 0)
        {
            fsm2.MakeChart(img_path, "png", true);
        }
        else if (sfd->FilterIndex == 1)
        {
            fsm2.MakeChart(img_path, "gif", true);
        }
        else
        {
            fsm2.MakeChart(img_path, "bmp", true);
        }
        Runtime::InteropServices::Marshal::FreeHGlobal(IntPtr(img_path));
    }

    sfd->Title = "Save FSM File";
```

```
        sfd->FileName = "";
        sfd->Filter = "FSM File (*.fsm)|*.fsm|All Files (*.*)|*.*";
        if (sfd->ShowDialog() == Windows::Forms::DialogResult::OK)
        {
            char* fsm_path = (char*)(void*)Runtime::InteropServices::Marshal
                ::StringToHGlobalAnsi(sfd->FileName);
            fsm2.WriteXml(fsm_path);
            Runtime::InteropServices::Marshal::FreeHGlobal(IntPtr(fsm_path));
        }
    }   // end synthesize
    else    // simulate
    {
        thesis_can::frmSimulate^ simPanel = gcnew thesis_can::frmSimulate;
        simPanel->ShowDialog();
    } // end simulate

    printf("Done\n");
    return 0;
}


void display_progress(float p, char* s)
{
    p *= 100;
    printf("%.1f %s\n", p, s);
}
```

A.1.2 globals.h

```
#pragma once
#include "CanFormat.h"

#define CAN_VIRTUAL 0
#define CAN_HARDWARE 1



struct PROGRESS_STRUCT
{
    float percentage;
    char description[128];
};
```

```
extern PROGRESS_STRUCT PROGRESS;



void UPDATE_PROGRESS(float p);

extern int CAN_MODE;

extern CanFormat can_dbc;
```

A.1.3 globals.cpp

```
#include "stdafx.h"
#include "globals.h"

PROGRESS_STRUCT PROGRESS;

void UPDATE_PROGRESS(float p)
{
    PROGRESS.percentage = p;
}

int CAN_MODE = CAN_VIRTUAL;


CanFormat can_dbc;
```

A.1.4 clg.h

```
#pragma once

/*
Clg File Specification

The clg file is a binary format consisting of a header followed by fixed length
    CAN records.

BYTE ORDERING:
All data in structures defined below is in little-endian (Intel) format.
```

```
HEADER:
The header is 276 bytes long and is defined as follows:
*/
#include "windows.h"

//typedef unsigned int  DWORD;      // a 32 bit unsigned value
//typedef unsigned char BYTE;       // an 8 bit unsigned value

// DataType and DataSubType:
#define DATA_TYPE_CAN               1

#define DATA_SUBTYPE_CAN11          11  // file contains only 11 bit data
#define DATA_SUBTYPE_CAN29          29  // file contains only 29 bit data
#define DATA_SUBTYPE_CAN_MIXED  20  // file contains both 11 and 29 bit data

#define CLG_FILE_ID                 '2glc'// becomes glc2 in file

typedef struct {
    DWORD   FileID;         // must be CLG_FILE_ID
    BYTE    ValidityCheck;  // must be '-'
    BYTE    DataType;       // for CAN must be DATA_TYPE_CAN
    BYTE    DataSubType;    // see DATA_SUBTYPE_ defines
    BYTE    Reserved;
    DWORD   NtpTimeSecs;    // future use (set to zero for now)
    DWORD   NtpTimeFracs;   // future use (set to zero for now)
    DWORD   RefStartTime;   // timeGetTime() value at start of log (see below)
    BYTE    HashTbl[256];   // see below
} CAN_FILE_HEADER2;


// Notes on CAN_FILE_HEADER2:
// RefStartTime (for use by other programs)
//          This is the time in milliseconds returned by timeGetTime() which
//          corresponds to a log time of zero.
// HashTbl
//          This is a bit map (2048 bits) indicating which ids are included in
//          the log. A '1' in a particular bit position indicates that the
//          corresponding CAN id is included.  Byte zero bit zero represents an
//          id of zero, byte zero bit 1 represents 1, byte 1 bit 0 represents 8.
//          For 29 bit ids first get the 11 bit equivalent before calculating the
//          hash table representation.  Use GET_11FROM29_BIT_ID(x) defined below.

// The following can be used to extract the center portion of a 29 bit id.
// The center portion is used in the hash table and in the can parser functions.
```

```
#define CAN_ID_MASK_11_BIT          0x7ff
#define GET_11FROM29_BIT_ID(x)  ((x&(CAN_ID_MASK_11_BIT<<13))>>13)
#define GET_EQIV_29FROM11_BIT(x)    (x<<13)


/*
CAN MESSAGES:
Following the header are CAN message records.  Each one is 16 bytes long.
Even though each CAN message contains a time stamp, the messages should
be ordered correctly in the file by time (i.e earlier CAN messages are
placed in the file before later CAN messages). Each message is defined as
follows:
*/

typedef struct {
    DWORD   MsecFromStart;  // relative time (millisecs) from start of CAN message log
    DWORD   id;             // This field represents more that just the CAN id.  It
                                // includes flags in the most significant bits and a
                                    //source
                                // bus id (see below for details).
    BYTE    data[8];            // actual CAN bytes (see also below)
} CAN_MSG2;

/*
Sub fields of the id field of CAN_MSG2:
*/

// CAN id                   this goes into the lowest 11 or 29 bits of the id field
#define CAN_ID_MASK_11_BIT      0x7ff
#define CAN_ID_MASK_29_BIT      0x1fffffff
#define UNIQUE_29BIT_CAN_BITS   (CAN_ID_MASK_29_BIT & (~(CAN_ID_MASK_11_BIT | SRC_BUS_MASK)))
// source bus number        this goes into the 3-bit field defined by SRC_BUS_MASK
//                              The source bus is only used with 11-bit CAN ids.  Each
//                              project has unique associations between this number and
//                              physical source bus.
#define SRC_BUS_MASK            0x00003800
#define SRC_BUS_SHIFT       11  // SRC_BUS_MASK = 7<<SRC_BUS_SHIFT
#define GET_SRC_BUS(id)     ((id & SRC_BUS_MASK)>>SRC_BUS_SHIFT)
#define CODE_SRC_BUS(src)   ((src<< SRC_BUS_SHIFT) & SRC_BUS_MASK)
#define MAX_SRC_BUS_NUM     7
// flags
#define SHORT_CAN_MSG_FLAG  0x80000000  // indicates fewer than 8 bytes in msg
                                        // (actual length is coded in data byte 7)
#define INFO_MSG_FLAG           0x40000000  // indicates not a CAN Id but an information
```

```
                                        // msg (data bytes give info)
/*
CAN data bytes:
The can data bytes are ordered in the order received from the bus (i.e. first byte
received goes into data[0], second byte goes into data[1]. If a CAN message contains
8 bytes then all 8 data byte are populated.  If a CAN message contains fewer than 8
bytes then the first bytes get the actural data and the last byte (data[7]) is set
to the number of message bytes.



VIDEO synchronization messages:
In order to synchronize video, pseodo-CAN messages are inserted into the CAN log
stream. The ids for these sync messages are project-specific but typically 0x1a1
has been used for the first video channel sync, 0x1a2 for the second, 0x1a3 for
the third etc

Each video sync message contains 8 bytes of data which are interpreted as follows:
Bytes 0 to 3: video frame number (in big endian format)
Bytes 4 to 7: may be set to zero or to frame time (in big endian format)


*/
```

A.1.5 List.h

```
#pragma once

typedef unsigned char BYTE;

template <class T>
struct LINK
{
    T* value;
    LINK<T>* next;
};

template <class T>
class LIST
{
public:
    LIST(void)
```

```
{
    head = 0;
    tail = 0;
    count = 0;
}

~LIST(void) { }


LINK<T>* Add(T* value)
{
    if (count == 0)
    {
        head = new LINK<T>;
        head->next = 0;
        head->value = value;
        tail = head;
        count = 1;
    }
    else
    {
        tail->next = new LINK<T>;
        tail = tail->next;
        tail->value = value;
        tail->next = 0;
        count++;
    }

    return tail;

}

LINK<T>* AddUnique(T* value, bool (*equals)(T*, T*))
{
    if (0 == Find(value, equals))
    {
        return Add(value);
    }
    else
    {
        return 0;
    }
}
```

```
LINK<T>* AddTop(T* value)
{
    if (count == 0)
    {
        return Add(value);
    }
    else
    {
        LINK<T>* n = new LINK<T>;
        n->next = head;
        n->value = value;
        head = n;
        count++;
        return n;
    }


}

LINK<T>* Seek(int index)
{
    if (index < 0) return NULL;
    if (index >= count) return NULL;

    LINK<T>* seeker = head;

    while (seeker != NULL && index > 0)
    {
        seeker = seeker->next;
        index--;
    }

    return seeker;
}

LINK<T>* Find(T* value, bool (*equals) (T*, T*) )
{
    LINK<T>* seeker = head;
    while (seeker != NULL)
    {
        if ( (equals) (seeker->value, value) )
        {
            return seeker;
        }
```

```
        seeker = seeker->next;
    }
    return NULL;
}

void Iterate(void (*action) (T*))
{
    LINK<T>* seeker = head;
    while (seeker != NULL)
    {
        (action) (seeker->value);
        seeker = seeker->next;
    }
}



void ClearAndDelete()
{
    LINK<T>* seeker = head;
    LINK<T>* temp;

    while (seeker != NULL)
    {
        temp = seeker;
        seeker = seeker->next;
        delete temp->value;
        delete temp;
    }

    head = 0;
    tail = 0;
    count = 0;
}

void Clear()
{
    LINK<T>* seeker = head;
    LINK<T>* temp;

    while (seeker != NULL)
    {
        temp = seeker;
        seeker = seeker->next;
```

```
            //delete temp->value;
            delete temp;
        }

        head = 0;
        tail = 0;
        count = 0;
    }

    LINK<T>* head;
    LINK<T>* tail;
    int count;

};
```

A.1.6 dl_list.h

```
#pragma once


template <class T>
struct DOUBLE_LINK
{
    T* value;
    DOUBLE_LINK<T>* next;
    DOUBLE_LINK<T>* prev;
};



template <class T>
class DL_LIST
{
public:
    DL_LIST(void)
    {
        head = 0;
        tail = 0;
        count = 0;
    }

    ~DL_LIST(void)
    {
```

```
}

DOUBLE_LINK<T>* AddCopy(T* value)
{
    T* x = new T;
    *x = *value;
    return Add(x);
}

DOUBLE_LINK<T>* Add(T* value)
{
    if (count == 0)
    {
        head = new DOUBLE_LINK<T>;
        head->next = 0;
        head->prev = 0;
        head->value = value;
        tail = head;
        count = 1;
    }
    else
    {
        tail->next = new DOUBLE_LINK<T>;
        tail->next->prev = tail;
        tail = tail->next;
        tail->value = value;
        tail->next = 0;
        count++;
    }

    return tail;

}

DOUBLE_LINK<T>* AddUniqueCopy(T* value, bool (*equals)(T*, T*))
{
    T* x = new T;
    *x = *value;
    return AddUnique(x);
}

DOUBLE_LINK<T>* AddUnique(T* value, bool (*equals)(T*, T*))
{
```

```
    DOUBLE_LINK<T>* result;
    result = Find(value, equals);

    if (0 == result)
    {
        result = Add(value);
    }

    return result;
}

DOUBLE_LINK<T>* AddFront(T* value)
{
    if (count == 0)
    {
        return Add(value);
    }
    else
    {
        DOUBLE_LINK<T>* n = new DOUBLE_LINK<T>;
        n->next = head;
        n->value = value;
        n->prev = 0;
        head = n;
        count++;
        return n;
    }


}


DOUBLE_LINK<T>* Get(int index)
{
    if (index < 0) return NULL;
    if (index >= count) return NULL;

    DOUBLE_LINK<T>* seeker = head;

    while (seeker != NULL && index > 0)
    {
        seeker = seeker->next;
        index--;
    }
```

```
    return seeker;
}

DOUBLE_LINK<T>* Find(T* value, bool (*equals) (T*, T*) )
{
    DOUBLE_LINK<T>* seeker = head;
    while (seeker != NULL)
    {
        if ( (equals) (seeker->value, value) )
        {
            return seeker;
        }
        seeker = seeker->next;
    }
    return NULL;
}

void Iterate(void (*action) (T*))
{
    DOUBLE_LINK<T>* seeker = head;
    while (seeker != NULL)
    {
        (action) (seeker->value);
        seeker = seeker->next;
    }
}



void ClearAndDelete()
{
    DOUBLE_LINK<T>* seeker = head;
    DOUBLE_LINK<T>* temp;

    while (seeker != NULL)
    {
        temp = seeker;
        seeker = seeker->next;
        delete temp->value;
        delete temp;
    }

    head = 0;
```

```
    tail = 0;
    count = 0;
}

void Clear()
{
    DOUBLE_LINK<T>* seeker = head;
    DOUBLE_LINK<T>* temp;

    while (seeker != NULL)
    {
        temp = seeker;
        seeker = seeker->next;
        //delete temp->value;
        delete temp;
    }

    head = 0;
    tail = 0;
    count = 0;
}

void Remove(DOUBLE_LINK<T>* x)
{
    if (head == x && tail == x)
    {
        head = 0;
        tail = 0;
        count = 0;
    }
    else if (head == x)
    {
        head = head->next;
        x->next->prev = 0;
        count--;
    }
    else if (tail == x)
    {
        tail = tail->prev;
        x->prev->next = 0;
        count--;
    }
    else
    {
```

```
        x->prev->next = x->next;
        x->next->prev = x->prev;
        count--;
    }

    delete x;
}

void RemoveAndDelete(DOUBLE_LINK<T>* x)
{
    if (head == x && tail == x)
    {
        head = 0;
        tail = 0;
        count = 0;
    }
    else if (head == x)
    {
        head = head->next;
        x->next->prev = 0;
        count--;
    }
    else if (tail == x)
    {
        tail = tail->prev;
        x->prev->next = 0;
        count--;
    }
    else
    {
        x->prev->next = x->next;
        x->next->prev = x->prev;
        count--;
    }

    delete x->value;
    delete x;

}




DOUBLE_LINK<T>* head;
```

```
    DOUBLE_LINK<T>* tail;
    int count;



};
```

A.2 Synthesis Code

A.2.1 FiniteStateMachine.h

```
#pragma once

#include "FsmState.h"
#include "CanFormat.h"
#include "clg.h"

class FiniteStateMachine
{
public:
    FiniteStateMachine(void);
    ~FiniteStateMachine(void);

    FsmState* FindState(int number);

    DL_LIST<FsmState> States;

    void Synthesize(char* clg_path, CanFormat* dbc, void(*progress)(float,char*));
    //void MakeChart(char* result_path, char* image_type = "png");

    void Print();

    void MakeChart(char* output_path, char* image_type = "png",
        bool remove_gv_file = true);

    FsmState* GetStateByNumber(int number);

    void WriteXml(char* path);
    void ReadXml(char* path);


private:
    // synthesis members
```

```
    void ProcessClgEvent(CanFormat* dbc, CAN_MSG2* message, DWORD& previousEventTime);
    void ReduceEquivalency();
    void RecycleDeadState();
    FsmState* currentSynthesisState;
    FsmState* SetNextState(FsmTransition* t);
    void CombineStates(DOUBLE_LINK<FsmState>* s1, DOUBLE_LINK<FsmState>* s2);
    void CopyTransitions(FsmState* from, FsmState* to);

};
```

A.2.2 FiniteStateMachine.cpp

```
#include "StdAfx.h"
#include "FiniteStateMachine.h"

FiniteStateMachine::FiniteStateMachine(void)
{
    FsmState new_state;
    new_state.Number = 1;
    States.AddCopy(&new_state);
    currentSynthesisState = States.head->value;
}

FiniteStateMachine::~FiniteStateMachine(void)
{
    States.ClearAndDelete();
}

FsmState* FiniteStateMachine::FindState(int number)
{
    for (DOUBLE_LINK<FsmState>* s = States.head; s != 0; s = s->next)
    {
        if (s->value->Number == number) return s->value;
    }
    return 0;
}

void FiniteStateMachine::Synthesize(char* clg_path, CanFormat* dbc, void(*progress)(float,char*))
{
    if (progress)
    {
        progress(0, "Processing events");
    }
```

```
// open file
FILE* fr = fopen(clg_path, "rb");


CAN_FILE_HEADER2 header;
CAN_MSG2 entry;

// find input file size
fseek(fr, 0, SEEK_END);
long in_size = ftell(fr);
fseek(fr, 0, SEEK_SET);

// calculate number of entries from file size
int num_entries = (in_size - sizeof(CAN_FILE_HEADER2)) / sizeof(CAN_MSG2);

// read header
fread( &header, sizeof(CAN_FILE_HEADER2), 1, fr );

int entry_num = 0;

DWORD ltime = 0;

while (!feof(fr))
{
    // read an entry from the input file
    fread( &entry, sizeof(CAN_MSG2), 1, fr );

    ProcessClgEvent(dbc, &entry, ltime);

    entry_num++;
    if (progress)
    {
        progress(((float)entry_num)/((float)num_entries), "Processing events");
    }
}

if (progress)
{
    progress(1.0, "Processing events complete");
}


// close files
```

```
    fclose(fr);

    progress(0.0f, "Equivalency reduction");
    ReduceEquivalency();
    progress(1.0f, "Equivalency reduction complete");

    progress(0.0f, "Recycling dead state");
    RecycleDeadState();
    progress(1.0f, "Recycling dead state complete");

}

void FiniteStateMachine::ProcessClgEvent(CanFormat* dbc, CAN_MSG2* message,
    DWORD& previousEventTime)
{
    message->id &= CAN_ID_MASK_11_BIT;

    MESSAGE* m = dbc->FindMessage(message->id);

    // id is irrelevant
    if (m == 0) return;

    // no change in data
    if (0 == memcmp(m->data, message->data, 8)) return;

    double old_value, new_value;
    unsigned char old_data[8];

    memcpy(old_data, m->data, 8);
    SIGNAL* s;

    //DL_LIST<FsmSignal> changedSignals;
    FsmSignal newSignal;

    FsmTransition* nt = new FsmTransition;

    for (LINK<SIGNAL>* sl = m->signals.head; sl != 0; sl = sl->next)
    {
        // for each signal, determine change and relevance
        s = sl->value;

        if (s->direction != SIGDIR_IGNORE)
        {
            memcpy(m->data, old_data, 8);
```

```
            old_value = s->Get();

            memcpy(m->data, message->data, 8);
            new_value = s->Get();

            if (old_value != new_value)
            {
                newSignal.value = new_value;
                strcpy(newSignal.name, s->name);
                if (0 == strcmp(s->type, "ENM"))
                {
                    s->FindEnumeration( (int)new_value, newSignal.enumeration );
                }

                nt->signals.AddCopy( &newSignal );
            }

        }

    }

    if (nt->signals.count > 0)
    {
        // add a transition using the signals found

        nt->count = 1;
        nt->time = message->MsecFromStart - previousEventTime;
        previousEventTime = message->MsecFromStart;

        if (m->send_enable)
        {
            nt->direction = DIRECTION_TX;
        }
        else
        {
            nt->direction = DIRECTION_RX;
        }

        nt->id = m->id;

        FsmState* nextState = SetNextState(nt);
        currentSynthesisState->AddTransition(nt);
        currentSynthesisState = nextState;
    }
```

```
    else
    {
        delete nt;
    }


}


void FiniteStateMachine::ReduceEquivalency()
{
    DOUBLE_LINK<FsmState>* s1;
    DOUBLE_LINK<FsmState>* s2;

    bool restart;


    /* remove */ Print(); /* remove */

    s1 = States.head;

    while (s1 != 0)
    {
        restart = false;
        s2 = s1->next;
        while (s2 != 0 && !restart)
        {
            if (s1->value->TransitionsEqual(s2->value))
            {
                printf("S%d = S%d \n", s1->value->Number, s2->value->Number);

                CombineStates(s1, s2);
                restart = true;

                /* remove */ Print(); /* remove */
            }
            s2 = s2->next;
        }

        if (restart)
        {
            s1 = States.head;
        }
        else
```

```
        {
            s1 = s1->next;
        }
    }
}


void FiniteStateMachine::CombineStates(DOUBLE_LINK<FsmState>* s1, DOUBLE_LINK<FsmState>* s2)
{
    // copy transitions from s2 to s1
    CopyTransitions(s2->value, s1->value);

    // remove s2
    int removal_state = s2->value->Number;
    States.RemoveAndDelete(s2);

    // for all transitions, if nextState = s2.state, now nextState = s1.state
    for (DOUBLE_LINK<FsmState>* state = States.head; state != 0; state = state->next)
    {
        for (DOUBLE_LINK<FsmTransition>* transition = state->value->Transitions.head;
            transition != 0; transition = transition->next)
        {
            if (transition->value->nextState == removal_state)
            {
                transition->value->nextState = s1->value->Number;
            }
        }

    }
}

void FiniteStateMachine::CopyTransitions(FsmState* from, FsmState* to)
{
    FsmTransition* target;
    //double result;
    //double result2;

    for (DOUBLE_LINK<FsmTransition>* t = from->Transitions.head; t != 0; t = t->next)
    {
        target = to->FindTransition(t->value, true);

        if (target != 0)
        {
            // update using values
```

```
            //int new_count = target->count + t->value->count;

            //target->time = (target->time / new_count) * target->count + (t->value->time /
                // new_count);

            //target->count = new_count;
            //result = (double)target->time;
            //result /= (double)(target->count + t->value->count);
            //result *= (double)target->count;
            //result2 = (double)t->value->time;
            //result2 /= (double)(target->count + t->value->count);
            //result2 *= (double)t->value->count;
            //target->time = (unsigned long)(result + result2);

            target->time = (target->time / (target->count + t->value->count) * target->count)
                + (t->value->time / (target->count + t->value->count) * t->value->count);
            target->count += t->value->count;

        }
        else
        {
            // states were not really equivalent
        }
    }
}


void FiniteStateMachine::RecycleDeadState()
{
    DOUBLE_LINK<FsmState>* dead_state = States.tail;
    if (dead_state->value->Transitions.count != 0) return;

    // redirect incoming transitions
    for (DOUBLE_LINK<FsmState>* state = States.head; state != 0; state = state->next)
    {
        for (DOUBLE_LINK<FsmTransition>* transition = state->value->Transitions.head;
            transition != 0; transition = transition->next)
        {
            if (transition->value->nextState == dead_state->value->Number)
            {
                transition->value->nextState = 1;
            }
        }
```

```
    }

    States.RemoveAndDelete(dead_state);
}


FsmState* FiniteStateMachine::SetNextState(FsmTransition* t)
{
    FsmTransition* found_transition;

    for (DOUBLE_LINK<FsmState>* state = States.head; state != 0; state = state->next)
    {
        found_transition = state->value->FindTransition(t, false);
        if (found_transition != 0)
        {
            t->nextState = found_transition->nextState;
            return FindState(t->nextState);
        }
    }

    FsmState* newState = new FsmState;
    newState->Number = States.tail->value->Number + 1;
    States.Add(newState);
    t->nextState = newState->Number;
    return newState;
}

void FiniteStateMachine::Print()
{
    // for each state
    for (DOUBLE_LINK<FsmState>* state = States.head; state != 0; state = state->next)
    {
        // print state number
        printf("S%d \n", state->value->Number);

        // for each transition
        for (DOUBLE_LINK<FsmTransition>* transition = state->value->Transitions.head;
            transition != 0; transition = transition->next)
        {
            // print count, time, id, direction, nextState
            printf("   %dx %dms 0x%x %d  S%d \n",
                transition->value->count, transition->value->time, transition->value->id,
                (int)transition->value->direction, transition->value->nextState);
```

```
            // for each signal
            for (DOUBLE_LINK<FsmSignal>* signal = transition->value->signals.head;
                signal != 0; signal = signal->next)
            {
                // print name,value
                printf("      %s = %g \n", signal->value->name, signal->value->value);
            }
        }
    }
}

void FiniteStateMachine::MakeChart(char* output_path, char* image_type, bool remove_gv_file)
{
    char gv_path[260];
    sprintf(gv_path, "%s.gv", output_path);

    FILE* gvf = fopen(gv_path, "w");

    // write header
    fprintf(gvf, "digraph fsm {\n");
    fprintf(gvf, "\tnode [shape=doublecircle];\n");
    fprintf(gvf, "\tnode [shape=circle];\n");

    char txrx;

    // write all transitions
    for (DOUBLE_LINK<FsmState>* state = States.head; state != 0; state = state->next)
    {
        for (DOUBLE_LINK<FsmTransition>* trans = state->value->Transitions.head; trans != 0;
            trans = trans->next)
        {
            if (trans->value->direction == DIRECTION_TX) txrx = 'T';
            else txrx = 'R';

            fprintf(gvf, "\t%d -> %d [label=\"[%c] ",
                state->value->Number, trans->value->nextState, txrx);

            for (DOUBLE_LINK<FsmSignal>* sig = trans->value->signals.head; sig != 0;
                sig = sig->next)
            {
                if (strlen(sig->value->enumeration) > 0)
                {
                    fprintf(gvf, "%s=%s", sig->value->name, sig->value->enumeration);
                }
```

```
                else
                {
                    fprintf(gvf, "%s=%g", sig->value->name, sig->value->value);
                }

                //if (sig->next != 0)
                {
                    fprintf(gvf, " ");
                }
            } // next signal

            fprintf(gvf, "(%d ms) (%dx)\", fontsize=8];\n", trans->value->time,
                trans->value->count);

        } // next transition

    } // next state


    // write footer
    fprintf(gvf, "}\n");

    fclose(gvf);

    // run graphvis
    char command[512];
    sprintf(command, "dot \"%s\" -T%s -o\"%s\"",
        gv_path, image_type, output_path );
    system(command);

    if (remove_gv_file)
    {
        remove(gv_path);
    }
}



FsmState* FiniteStateMachine::GetStateByNumber(int number)
{
    for (DOUBLE_LINK<FsmState>* s = States.head; s!=0; s = s->next)
    {
        if (s->value->Number == number)
        {
```

```
            return s->value;
        }
    }
    return 0;
}
```

```
void FiniteStateMachine::WriteXml(char* path)
{
    System::Xml::XmlWriterSettings^ settings = gcnew System::Xml::XmlWriterSettings();
    settings->Indent = true;
    //settings->IndentChars = "   ";

    System::Xml::XmlWriter^ writer = System::Xml::XmlWriter::Create(gcnew System::String(path),
        settings);

    writer->WriteStartElement("fsm");

    for (DOUBLE_LINK<FsmState>* statelink = States.head; statelink != 0;
        statelink = statelink->next)
    {
        writer->WriteStartElement("state");
        writer->WriteAttributeString("number", System::Convert
            ::ToString(statelink->value->Number));

        for (DOUBLE_LINK<FsmTransition>* translink = statelink->value->Transitions.head;
            translink != 0; translink = translink->next)
        {
            writer->WriteStartElement("transition");
            writer->WriteAttributeString("count", System::Convert
                ::ToString(translink->value->count));
            writer->WriteAttributeString("direction", System::Convert
                ::ToString((int)translink->value->direction));
            writer->WriteAttributeString("id", System::Convert
                ::ToString(translink->value->id));
            writer->WriteAttributeString("nextstate", System::Convert
                ::ToString(translink->value->nextState));
            writer->WriteAttributeString("time", System::String
```

```
                ::Format("{0}", translink->value->time));
            for (DOUBLE_LINK<FsmSignal>* siglink = translink->value->signals.head;
                siglink != 0; siglink = siglink->next)
            {
                writer->WriteStartElement("signal");
                writer->WriteAttributeString("name", gcnew System
                    ::String(siglink->value->name));
                writer->WriteAttributeString("value", System::String
                    ::Format("{0}", siglink->value->value));
                writer->WriteAttributeString("enumeration", gcnew System
                    ::String(siglink->value->enumeration));
                writer->WriteEndElement(); // signal
            }
            writer->WriteEndElement(); // transition

        }
        writer->WriteEndElement(); // state

    }


    writer->WriteEndElement(); // fsm
    writer->Flush();

    writer->Close();

}




void FiniteStateMachine::ReadXml(char* path)
{
    States.ClearAndDelete();

    System::Xml::XmlReader^ reader = System::Xml::XmlReader::Create(gcnew System::String(path));

    FsmSignal* currentSignal = 0;
    FsmTransition* currentTransition = 0;
    FsmState* currentState = 0;
    char* cptr;
```

```
while (reader->Read())
{
    if (reader->NodeType == System::Xml::XmlNodeType::Element)
    {
        if (reader->Name == "state")
        {
            currentState = new FsmState;
            States.Add(currentState);

            reader->MoveToAttribute("number");
            currentState->Number = System::Convert::ToInt32(reader->Value);
        }
        else if (reader->Name == "transition")
        {
            currentTransition = new FsmTransition;
            currentState->Transitions.Add(currentTransition);

            reader->MoveToAttribute("count");
            currentTransition->count = System::Convert::ToInt32(reader->Value);

            reader->MoveToAttribute("direction");
            currentTransition->direction = (FsmTransitionDirection)System::Convert
                ::ToInt32(reader->Value);

            reader->MoveToAttribute("id");
            currentTransition->id = System::Convert::ToInt32(reader->Value);

            reader->MoveToAttribute("nextstate");
            currentTransition->nextState = System::Convert::ToInt32(reader->Value);

            reader->MoveToAttribute("time");
            currentTransition->time = System::Convert::ToUInt32(reader->Value);

        }
        else if (reader->Name == "signal")
        {
            currentSignal = new FsmSignal;
            currentTransition->signals.Add(currentSignal);

            reader->MoveToAttribute("name");
            cptr = (char*)(void*)System::Runtime::InteropServices::Marshal
                ::StringToHGlobalAnsi(reader->Value);
            strcpy(currentSignal->name, cptr);
            System::Runtime::InteropServices::Marshal::FreeHGlobal(System::IntPtr(cptr));
```

```
                reader->MoveToAttribute("value");
                currentSignal->value = System::Convert::ToDouble(reader->Value);

                reader->MoveToAttribute("enumeration");
                cptr = (char*)(void*)System::Runtime::InteropServices::Marshal
                    ::StringToHGlobalAnsi(reader->Value);
                strcpy(currentSignal->enumeration, cptr);
                System::Runtime::InteropServices::Marshal::FreeHGlobal(System::IntPtr(cptr));

            }
        }

    }

    reader->Close();

}
```

A.2.3 FsmState.h

```
#pragma once

#include "FsmTransition.h"

class FsmState
{
public:
    FsmState(void);
    ~FsmState(void);

    bool TransitionsEqual(FsmState* x);
    void AddTransition(FsmTransition* x);
    FsmTransition* FindTransition(FsmTransition* x, bool matchNextStates);

    int Number;
    DL_LIST<FsmTransition> Transitions;
};
```

A.2.4 FsmState.cpp

```
#include "StdAfx.h"
#include "FsmState.h"

FsmState::FsmState(void)
{
    Number = 0;
}

FsmState::~FsmState(void)
{
    Transitions.ClearAndDelete();
}

bool FsmState::TransitionsEqual(FsmState* x)
{
    if (x->Transitions.count != Transitions.count) return false;

    bool matched;

    for (DOUBLE_LINK<FsmTransition>* t1 = x->Transitions.head; t1 != 0; t1 = t1->next)
    {
        matched = false;
        for (DOUBLE_LINK<FsmTransition>* t2 = Transitions.head; (t2 != 0) && !matched;
            t2 = t2->next)
        {
            if (t1->value->Equals(t2->value)) matched = true;
        }
        if (!matched) return false;
    }

    return true;
}

void FsmState::AddTransition(FsmTransition* x)
{
    FsmTransition* t = FindTransition(x, true);

    if (t != 0)
    {
        // update t using values from x, then return
        int new_count = t->count + x->count;
```

```
        t->time = (t->time / new_count) * t->count + (x->time / new_count);

        t->count = new_count;

        delete x;
    }
    else
    {
        // add new transition
        Transitions.Add(x);
    }
}

FsmTransition* FsmState::FindTransition(FsmTransition* x, bool matchNextStates)
{
    for (DOUBLE_LINK<FsmTransition>* t = Transitions.head; t != 0; t = t->next)
    {

        if (matchNextStates)
        {
            if (x->Equals(t->value)) return t->value;
        }
        else
        {
            if (x->SignalsEqual(t->value)) return t->value;
        }
    }

    return 0;
}
```

A.2.5 FsmTransition.h

```
#pragma once

#include "DL_LIST.h"
#include "FsmSignal.h"

enum FsmTransitionDirection { DIRECTION_TX, DIRECTION_RX };

class FsmTransition
{
public:
```

```
    FsmTransition(void);
    ~FsmTransition(void);

    bool SignalsEqual(FsmTransition* x);
    bool Equals(FsmTransition* x);

    int count;
    unsigned long time;
    int id;
    FsmTransitionDirection direction;
    int nextState;
    DL_LIST<FsmSignal> signals;
};
```

A.2.6 FsmTransition.cpp

```
#include "StdAfx.h"
#include "FsmTransition.h"

FsmTransition::FsmTransition(void)
{
    count = 0;
    time = 0;
    id = 0;
    direction = DIRECTION_TX;
    nextState = 0;
}

FsmTransition::~FsmTransition(void)
{
    signals.ClearAndDelete();
}

bool FsmTransition::SignalsEqual(FsmTransition* x)
{
    if (x->signals.count != signals.count) return false;

    bool matched;

    for (DOUBLE_LINK<FsmSignal>* s1 = x->signals.head; s1 != 0; s1 = s1->next)
    {
        matched = false;
```

```
        for (DOUBLE_LINK<FsmSignal>* s2 = signals.head; (s2 != 0) && (!matched); s2 = s2->next)
        {
            if (s1->value->Equals(s2->value)) matched = true;
        }

        if (!matched) return false;

    }

    return true;
}

bool FsmTransition::Equals(FsmTransition* x)
{
    if (x->id != id) return false;
    if (x->direction != direction) return false;
    if (x->nextState != nextState) return false;
    return SignalsEqual(x);
}
```

A.2.7 FsmSignal.h

```
#pragma once

class FsmSignal
{
public:
    FsmSignal(void);
    ~FsmSignal(void);

    bool Equals(FsmSignal* x);

    char name[100];
    double value;
    char enumeration[100];

};
```

A.2.8 FsmSignal.cpp

```
#include "StdAfx.h"
#include "FsmSignal.h"
```

```
#include "string.h"

FsmSignal::FsmSignal(void)
{
    strcpy(name, "");
    value = 0;
    strcpy(enumeration, "");
}

FsmSignal::~FsmSignal(void)
{
}

bool FsmSignal::Equals(FsmSignal* x)
{
    if (x->value != value) return false;
    if (0 != strcmp(x->name, name)) return false;
    return true;
}
```

A.2.9 frmSignalSorting.h

```
#pragma once

#include "globals.h"

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;


namespace thesis_can {

    /// <summary>
    /// Summary for frmSignalSorting
    ///
    /// WARNING: If you change the name of this class, you will need to change the
    ///          'Resource File Name' property for the managed resource compiler tool
    ///          associated with all .resx files this class depends on.  Otherwise,
```

```
    ///          the designers will not be able to interact properly with localized
    ///          resources associated with this form.
    /// </summary>
    public ref class frmSignalSorting : public System::Windows::Forms::Form
    {
    public:
        frmSignalSorting(void)
        {
            InitializeComponent();
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~frmSignalSorting()
        {
            if (components)
            {
                delete components;
            }
        }
    private: System::Windows::Forms::ListView^  listView1;
    private: System::Windows::Forms::ColumnHeader^  columnHeader1;
    private: System::Windows::Forms::ColumnHeader^  columnHeader2;
    private: System::Windows::Forms::ColumnHeader^  columnHeader3;
    private: System::Windows::Forms::Button^  button1;
    private: System::Windows::Forms::Button^  button2;
    private: System::Windows::Forms::Button^  button3;
    private: System::Windows::Forms::Button^  button4;
    protected:

    private:
        /// <summary>
        /// Required designer variable.
        /// </summary>
        System::ComponentModel::Container ^components;

#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
```

```
/// the contents of this method with the code editor.
/// </summary>
void InitializeComponent(void)
{
    this->listView1 = (gcnew System::Windows::Forms::ListView());
    this->columnHeader1 = (gcnew System::Windows::Forms::ColumnHeader());
    this->columnHeader2 = (gcnew System::Windows::Forms::ColumnHeader());
    this->columnHeader3 = (gcnew System::Windows::Forms::ColumnHeader());
    this->button1 = (gcnew System::Windows::Forms::Button());
    this->button2 = (gcnew System::Windows::Forms::Button());
    this->button3 = (gcnew System::Windows::Forms::Button());
    this->button4 = (gcnew System::Windows::Forms::Button());
    this->SuspendLayout();
    //
    // listView1
    //
    this->listView1->Columns->AddRange(gcnew cli::array< System::Windows::Forms
        ::ColumnHeader^  >(3) {this->columnHeader1, this->columnHeader2,
        this->columnHeader3});
    this->listView1->FullRowSelect = true;
    this->listView1->Location = System::Drawing::Point(12, 12);
    this->listView1->Name = L"listView1";
    this->listView1->Size = System::Drawing::Size(541, 478);
    this->listView1->TabIndex = 0;
    this->listView1->UseCompatibleStateImageBehavior = false;
    this->listView1->View = System::Windows::Forms::View::Details;
    //
    // columnHeader1
    //
    this->columnHeader1->Text = L"ID";
    this->columnHeader1->Width = 91;
    //
    // columnHeader2
    //
    this->columnHeader2->Text = L"Signal";
    this->columnHeader2->Width = 320;
    //
    // columnHeader3
    //
    this->columnHeader3->Text = L"Type";
    this->columnHeader3->Width = 103;
    //
    // button1
    //
```

```
this->button1->Location = System::Drawing::Point(12, 496);
this->button1->Name = L"button1";
this->button1->Size = System::Drawing::Size(75, 23);
this->button1->TabIndex = 1;
this->button1->Text = L"Ignore";
this->button1->UseVisualStyleBackColor = true;
this->button1->Click += gcnew System::EventHandler(this, &frmSignalSorting
    ::button1_Click);
//
// button2
//
this->button2->Location = System::Drawing::Point(93, 496);
this->button2->Name = L"button2";
this->button2->Size = System::Drawing::Size(75, 23);
this->button2->TabIndex = 2;
this->button2->Text = L"Receive";
this->button2->UseVisualStyleBackColor = true;
this->button2->Click += gcnew System::EventHandler(this, &frmSignalSorting
    ::button2_Click);
//
// button3
//
this->button3->Location = System::Drawing::Point(174, 496);
this->button3->Name = L"button3";
this->button3->Size = System::Drawing::Size(75, 23);
this->button3->TabIndex = 3;
this->button3->Text = L"Transmit";
this->button3->UseVisualStyleBackColor = true;
this->button3->Click += gcnew System::EventHandler(this, &frmSignalSorting
    ::button3_Click);
//
// button4
//
this->button4->Location = System::Drawing::Point(478, 496);
this->button4->Name = L"button4";
this->button4->Size = System::Drawing::Size(75, 23);
this->button4->TabIndex = 4;
this->button4->Text = L"OK";
this->button4->UseVisualStyleBackColor = true;
this->button4->Click += gcnew System::EventHandler(this, &frmSignalSorting
    ::button4_Click);
//
// frmSignalSorting
//
```

```
            this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(565, 531);
            this->Controls->Add(this->button4);
            this->Controls->Add(this->button3);
            this->Controls->Add(this->button2);
            this->Controls->Add(this->button1);
            this->Controls->Add(this->listView1);
            this->FormBorderStyle = System::Windows::Forms::FormBorderStyle
                ::FixedSingle;
            this->MaximizeBox = false;
            this->MinimizeBox = false;
            this->Name = L"frmSignalSorting";
            this->Text = L"Sort Signals";
            this->Load += gcnew System::EventHandler(this, &frmSignalSorting
                ::frmSignalSorting_Load);
            this->ResumeLayout(false);

        }
#pragma endregion
private: System::Void frmSignalSorting_Load(System::Object^  sender,
             System::EventArgs^  e)
{
    LINK<SIGNAL>* s;
    LINK<MESSAGE>* m = can_dbc.messages.head;
    while (m != 0)
    {
        s = m->value->signals.head;
        while (s != 0)
        {
            listView1->Items->Add(String::Format("{0:0}",m->value->id));
            listView1->Items[listView1->Items->Count-1]->SubItems
                ->Add(gcnew String(s->value->name));
            listView1->Items[listView1->Items->Count-1]->SubItems
                ->Add(directionString(s->value->direction));

            s = s->next;
        }

        m = m->next;
    }
}

private: String^ directionString(int dir)
```

```
{
    if (dir == SIGDIR_RECEIVE)
        return "Receive";
    else if (dir == SIGDIR_TRANSMIT)
        return "Transmit";
    else
        return "Ignore";
}

private: void changeSignalValues(int new_direction)
{
    char* name;
    int id;
    MESSAGE* m;

    for each (Windows::Forms::ListViewItem^ item in listView1->SelectedItems)
    {
        item->SubItems[2]->Text = directionString(new_direction);
        id = Convert::ToInt32(item->Text);
        name = (char*)(void*)Runtime::InteropServices::Marshal
            ::StringToHGlobalAnsi(item->SubItems[1]->Text);
        m = can_dbc.FindMessage(id);
        m->FindSignal(name)->direction = new_direction;
        m->send_enable = (new_direction == SIGDIR_TRANSMIT);

        Runtime::InteropServices::Marshal::FreeHGlobal(IntPtr(name));
    }
}

private: System::Void button1_Click(System::Object^  sender, System::EventArgs^  e)
{
    changeSignalValues(SIGDIR_IGNORE);
    listView1->Focus();
}

private: System::Void button2_Click(System::Object^  sender, System::EventArgs^  e)
{
    changeSignalValues(SIGDIR_RECEIVE);
    listView1->Focus();
}

private: System::Void button3_Click(System::Object^  sender, System::EventArgs^  e)
{
    changeSignalValues(SIGDIR_TRANSMIT);
```

```
    listView1->Focus();
}

private: System::Void button4_Click(System::Object^  sender, System::EventArgs^  e)
{
     this->DialogResult = Windows::Forms::DialogResult::OK;
     this->Close();
}

}; // end form
}
```

A.3 Simulation Code

A.3.1 FsmSimulation.h

```
#pragma once

#include "FiniteStateMachine.h"

namespace SimModes {
    enum SimModes {
        Start, Complete, Error, Load, ChooseSendable, ChooseAny, WaitForReceive,
        WaitForSend, WaitForAny, Send
    };
}

class FsmSimulation
{
public:
    FsmSimulation(void);
    ~FsmSimulation(void);

    bool Done();
    void Service();
    //void Receive(int id, unsigned char data[]);
    void PrintStatus();

    CanFormat* dbc;
    FiniteStateMachine* fsm;

    void AdjustDbc();
```

```
    int GetCurrentState();
    void GetMode(char* result);

private:

    SimModes::SimModes mode;
    SimModes::SimModes oldMode;
    int nextState;
    FsmState* currentState;
    DWORD stateStartTime;
    bool stateHasTx;
    bool stateHasRx;
    FsmTransition* chosenTransition;
    FsmTransition* receivedTransition;
    //FsmReceiveQueue queue;

    void updateStateProperties();
    FsmTransition* GetRandomTransition();
    FsmTransition* ChooseWeightedRandom(DL_LIST<FsmTransition>* candidates);
    FsmTransition* CheckCanForTransition();
    void SendTransition(FsmTransition* t);
    bool CriteriaMet(FsmTransition* t);
};
```

A.3.2 FsmSimulation.cpp

```
#include "StdAfx.h"
#include "FsmSimulation.h"

FsmSimulation::FsmSimulation(void)
{
    mode = SimModes::Start;
    dbc = 0;
    fsm = 0;
    nextState = 1;
    currentState = 0;
    oldMode = SimModes::Complete;
}

FsmSimulation::~FsmSimulation(void)
{
}
```

```
void FsmSimulation::PrintStatus()
{
    if (oldMode == mode) return;
    oldMode = mode;

    if (currentState == 0) return;

    char description[128];

    switch (mode)
    {
        case SimModes::ChooseAny:
            strcpy(description, "Choosing between sendable and receivable");
        break;

        case SimModes::ChooseSendable:
            strcpy(description, "Choosing a send");
        break;

        case SimModes::Complete:
            strcpy(description, "Finished");
        break;

        case SimModes::Error:
            strcpy(description, "Error occurred");
        break;

        case SimModes::Load:
            strcpy(description, "Loading state");
        break;

        case SimModes::Send:
            strcpy(description, "Sending");
        break;

        case SimModes::Start:
            strcpy(description, "Initializing");
        break;

        case SimModes::WaitForAny:
            strcpy(description, "Waiting for chosen send or receive");
        break;
```

```
        case SimModes::WaitForReceive:
            strcpy(description, "Receiving");
        break;

        case SimModes::WaitForSend:
            strcpy(description, "Waiting to send");
        break;


    }



    printf("State=%d, Mode=%s\n", currentState->Number, description);
}


bool FsmSimulation::Done()
{
    return (mode == SimModes::Complete) || (mode == SimModes::Error);
}

void FsmSimulation::Service()
{
    switch (mode)
    {
        ////////////////////////////////////////////////////
        case SimModes::Start:
            nextState = 1;
            mode = SimModes::Load;
        break;
        ////////////////////////////////////////////////////
        case SimModes::Load:

            stateStartTime = GetTickCount();
            currentState = fsm->GetStateByNumber(nextState);
            //queue.Clear();

            if (currentState == 0)
            {
                mode = SimModes::Error;
            }
            else if (currentState->Transitions.count == 0)
            {
```

```
        mode = SimModes::Complete;
    }
    else
    {
        updateStateProperties();

        if (stateHasTx && stateHasRx)
        {
            mode = SimModes::ChooseAny;
        }
        else if (stateHasTx)
        {
            mode = SimModes::ChooseSendable;
        }
        else
        {
            mode = SimModes::WaitForReceive;
        }
    }

break;
//////////////////////////////////////////////////////
case SimModes::ChooseSendable:

    chosenTransition = GetRandomTransition();

    if (chosenTransition == 0)
    {
        mode = SimModes::Complete;
    }
    else
    {
        mode = SimModes::WaitForSend;
    }

break;
//////////////////////////////////////////////////////
case SimModes::ChooseAny:

    chosenTransition = GetRandomTransition();
    mode = SimModes::WaitForAny;

break;
//////////////////////////////////////////////////////
```

```
case SimModes::WaitForReceive:

    chosenTransition = CheckCanForTransition();
    if (chosenTransition != 0)
    {
        nextState = chosenTransition->nextState;
        mode = SimModes::Load;
    }

break;
//////////////////////////////////////////////////////
case SimModes::WaitForSend:

    if ( (GetTickCount() - stateStartTime) >= chosenTransition->time)
    {
        mode = SimModes::Send;
    }

break;
//////////////////////////////////////////////////////
case SimModes::WaitForAny:

    if ( (GetTickCount() - stateStartTime) >= chosenTransition->time )
    {
        if (chosenTransition->direction == DIRECTION_TX)
        {
            mode = SimModes::Send;
        }
        else
        {
            mode = SimModes::ChooseAny;
        }
    }
    else
    {
        receivedTransition = CheckCanForTransition();

        if (receivedTransition != 0)
        {
            nextState = receivedTransition->nextState;
            mode = SimModes::Load;
        }
    }
```

```
        break;
        ////////////////////////////////////////////////////
        case SimModes::Send:

            SendTransition(chosenTransition);
            nextState = chosenTransition->nextState;
            mode = SimModes::Load;

        break;
        ////////////////////////////////////////////////////
        case SimModes::Error:

            mode = SimModes::Start;

        break;
        ////////////////////////////////////////////////////
        case SimModes::Complete:
        break;
        ////////////////////////////////////////////////////
        default:

            mode = SimModes::Start;

        break;
    }
}

void FsmSimulation::SendTransition(FsmTransition* t)
{
    SIGNAL* sig;
    for (DOUBLE_LINK<FsmSignal>* sigLink = t->signals.head; sigLink != 0;
        sigLink = sigLink->next)
    {
        sig = dbc->FindSignal(t->id, sigLink->value->name);
        if (sig != 0)
        {
            sig->Set(sigLink->value->value);
        }
    }
}

//void FsmSimulation::Receive(int id, unsigned char data[])
//{
//  //queue.Add(id, data);
```

```
//}

FsmTransition* FsmSimulation::CheckCanForTransition()
{
    FsmTransition* result = 0;

    for (DOUBLE_LINK<FsmTransition>* t = currentState->Transitions.head; t != 0; t = t->next)
    {
        // if transition met and has the highest number of signals, it's the best transition
        if ((t->value->direction == DIRECTION_RX) && ((result == 0)
            || (t->value->signals.count > result->signals.count)) && CriteriaMet(t->value))
        {
            result = t->value;
        }
    }

    return result;

    //return queue.CheckForTransition(dbc, currentState);
}

bool FsmSimulation::CriteriaMet(FsmTransition* t)
{
    for (DOUBLE_LINK<FsmSignal>* fsig = t->signals.head; fsig != 0; fsig = fsig->next)
    {
        SIGNAL* dsig = dbc->FindSignal(t->id, fsig->value->name);
        if ( (dsig == 0) || (dsig->Get() != fsig->value->value) )
        {
            return false;
        }
        else
        {
            //printf("Transition (%s=%g) Received (%s=%g)\n",
            //  fsig->value->name, fsig->value->value,
            //  dsig->name, dsig->Get());
        }
    }

    return true;
}

void FsmSimulation::updateStateProperties()
{
    stateHasTx = false;
```

```
    stateHasRx = false;

    for (DOUBLE_LINK<FsmTransition>* t = currentState->Transitions.head; t!=0; t=t->next)
    {
        if (t->value->direction == DIRECTION_RX)
        {
            stateHasRx = true;
        }
        else
        {
            stateHasTx = true;
        }

        if (stateHasTx && stateHasRx)
        {
            return;
        }
    }
}

FsmTransition* FsmSimulation::GetRandomTransition()
{
    // no events
    if (currentState->Transitions.count == 0)
    {
        return 0;
    }

    // only one event
    if (currentState->Transitions.count == 1)
    {
        return currentState->Transitions.head->value;
    }

    // first try ignoring any transitions that are already set
    DL_LIST<FsmTransition> candidateTransitions;
    for (DOUBLE_LINK<FsmTransition>* transition = currentState->Transitions.head;
        transition != 0; transition = transition->next)
    {
        bool addCandidate = true;
        for (DOUBLE_LINK<FsmSignal>* sig = transition->value->signals.head; sig != 0
            && addCandidate; sig = sig->next)
        {
            SIGNAL* s = dbc->FindSignal(transition->value->id, sig->value->name);
```

```
            if ( (s == 0) || (s->Get() == sig->value->value) )
            {
                addCandidate = false;
            }
        }
        if (addCandidate)
        {
            candidateTransitions.Add( transition->value );
        }
    }

    FsmTransition* result = 0;

    if (candidateTransitions.count == 0)
    {
        result = ChooseWeightedRandom( &(currentState->Transitions) );
    }
    else
    {
        result = ChooseWeightedRandom( &candidateTransitions );
    }

    candidateTransitions.Clear();

    return result;
}

FsmTransition* FsmSimulation::ChooseWeightedRandom(DL_LIST<FsmTransition>* candidates)
{
    if (candidates->count == 0)
    {
        return 0;
    }
    else if (candidates->count == 1)
    {
        return candidates->head->value;
    }

    // sum the counts to find max value of random number
    int count_sum = 0;

    DOUBLE_LINK<FsmTransition>* iter;
    for (iter = candidates->head; iter != 0; iter = iter->next)
    {
```

```
        count_sum += iter->value->count;
    }


    srand(GetTickCount());
    int random = rand() % count_sum + 1;

    for (iter = candidates->head; iter != 0; iter = iter->next)
    {
        random -= iter->value->count;

        if (random <= 0)
        {
            return iter->value;
        }
    }

    return 0;
}

void FsmSimulation::AdjustDbc()
{
    // set all transmittable id's

    for (DOUBLE_LINK<FsmState>* state = fsm->States.head; state != 0; state = state->next)
    {
        for (DOUBLE_LINK<FsmTransition>* tr = state->value->Transitions.head; tr != 0;
            tr = tr->next)
        {
            MESSAGE* m = dbc->FindMessage(tr->value->id);
            if ((m != 0) && (tr->value->direction == DIRECTION_TX))
            {
                m->send_enable = 1;
            }
        }
    }
}

int FsmSimulation::GetCurrentState()
{
    if (currentState)
    {
        return currentState->Number;
    }
```

```
    else
    {
        return 0;
    }
}

void FsmSimulation::GetMode(char* result)
{
    switch (mode)
    {
        case SimModes::ChooseAny:
            strcpy(result, "Choosing between sendable and receivable");
        break;

        case SimModes::ChooseSendable:
            strcpy(result, "Choosing a send");
        break;

        case SimModes::Complete:
            strcpy(result, "Finished");
        break;

        case SimModes::Error:
            strcpy(result, "Error occurred");
        break;

        case SimModes::Load:
            strcpy(result, "Loading state");
        break;

        case SimModes::Send:
            strcpy(result, "Sending");
        break;

        case SimModes::Start:
            strcpy(result, "Initializing");
        break;

        case SimModes::WaitForAny:
            strcpy(result, "Waiting for chosen send or receive");
        break;

        case SimModes::WaitForReceive:
            strcpy(result, "Receiving");
```

```
        break;

        case SimModes::WaitForSend:
            strcpy(result, "Waiting to send");
        break;

        default:
            strcpy(result, "");
        break;
    }
}
```

A.3.3 FsmReceiveQueue.h

```
#pragma once

#include "FiniteStateMachine.h"

struct FsmPacket {
    int id;
    unsigned char data[8];
};

class FsmReceiveQueue
{
public:
    FsmReceiveQueue(void);
    ~FsmReceiveQueue(void);

    void Add(int id, unsigned char data[]);
    FsmTransition* CheckForTransition(CanFormat* dbc, FsmState* state);
    int GetSize();
    void Clear();

private:
    DL_LIST<FsmPacket> packets;

    bool AllCriteriaMet(FsmTransition* transition, MESSAGE* message);
    FsmTransition* FindBestTransition(FsmState* state, MESSAGE* message);

};
```

A.3.4 FsmReceiveQueue.cpp

```
#include "StdAfx.h"
#include "FsmReceiveQueue.h"

FsmReceiveQueue::FsmReceiveQueue(void)
{
}

FsmReceiveQueue::~FsmReceiveQueue(void)
{
}

void FsmReceiveQueue::Add(int id, unsigned char data[])
{
    bool add = true;
    // find the last entry of id
    DOUBLE_LINK<FsmPacket>* p = packets.tail;
    while (p != 0)
    {
        if (p->value->id == id)
        {
            if (0 == memcmp(p->value->data, data, 8))
            {
                add = false;
            }

            p = 0;
        }
        else
        {
            p = p->prev;
        }
    }

    if (add)
    {
        FsmPacket* fp = new FsmPacket;
        fp->id = id;
        memcpy(fp->data, data, 8);
        packets.Add(fp);
    }
}
```

```
FsmTransition* FsmReceiveQueue::CheckForTransition(CanFormat* dbc, FsmState* state)
{
    FsmPacket* packet;

    FsmTransition* result = 0;

    MESSAGE* message;

    unsigned char temp[8];

    while (packets.count > 0 && result == 0)
    {
        packet = packets.head->value;

        message = dbc->FindMessage(packet->id);
        if (message != 0)
        {
            memcpy(temp, message->data, 8);
            memcpy(message->data, packet->data, 8);

            result = FindBestTransition(state, message);

            memcpy(message->data, temp, 8);
        }

        packets.RemoveAndDelete(packets.head);
    }


    return result;
}


FsmTransition* FsmReceiveQueue::FindBestTransition(FsmState* state, MESSAGE* message)
{
    LIST<FsmTransition> candidates;

    for (DOUBLE_LINK<FsmTransition>* transition = state->Transitions.head; transition != 0;
        transition = transition->next)
    {
        if (AllCriteriaMet(transition->value, message))
        {
            candidates.Add(transition->value);
        }
```

```
    }

    // find the candidate transition with the highest number of signals
    FsmTransition* result = 0;
    if (candidates.head != 0)
    {
        result = candidates.head->value;
    }

    for (LINK<FsmTransition>* candidate = candidates.head; candidate != 0;
        candidate = candidate->next)
    {
        if (candidate->value->signals.count > result->signals.count)
        {
            result = candidate->value;
        }
    }

    candidates.Clear();

    return result;
}


bool FsmReceiveQueue::AllCriteriaMet(FsmTransition* transition, MESSAGE* message)
{
    if (message->id != transition->id) return false;

    SIGNAL* sig;

    for (DOUBLE_LINK<FsmSignal>* tsig = transition->signals.head; tsig != 0;
        tsig = tsig->next)
    {
        sig = message->FindSignal(tsig->value->name);
        if (sig == 0)
        {
            return false;
        }
        else if (sig->Get() != tsig->value->value)
        {
            return false;
        }
    }
```

```
    return true;
}

int FsmReceiveQueue::GetSize()
{
    return packets.count;
}

void FsmReceiveQueue::Clear()
{
    packets.ClearAndDelete();
}
```

A.3.5 CanFormat.h

```
#pragma once

#include "Message.h"

class CanFormat
{
public:
    CanFormat();
    ~CanFormat();

    LIST<MESSAGE> messages;

    void ReadDbcFile(char* path);

    MESSAGE* FindMessage(int id);
    MESSAGE* FindMessage(const char* name);
    SIGNAL* FindSignal(int msg_id, const char* sig_name);
    SIGNAL* FindSignal(const char* msg_name, const char* sig_name);
    SIGNAL* FindSignal(const char* sig_name);

    bool Receive(int id, BYTE data[]);

    MESSAGE* GetNextSendable();

    void print();

    void OpenLogFile(char* path);
    MESSAGE* ReadNext();
```

```
    void CloseLogFile();
    bool LogFileFinished();
    void PrintLogFile(char* path);

    void ResetData();

private:
    MESSAGE* AddMessage(char* current_line);
    void AddSignal(MESSAGE* m, char* current_line);
    void AddSignalComment(char* current_line);
    void AddSendInterval(char* current_line);
    void AddSignalType(char* current_line);
    void AddSignalLongName(char* current_line);
    void AddEnums(char* current_line);

    bool LineStartsWith(char* line, const char test[]);

    FILE* log_file;


};
```

A.3.6 CanFormat.cpp

```
#include "stdafx.h"
#include "CanFormat.h"

#include <stdio.h>


CanFormat::CanFormat()
{

}

CanFormat::~CanFormat()
{

}

void CanFormat::ReadDbcFile(char* path)
{
```

```
FILE* f = fopen(path, "r");

char current_line[500];

MESSAGE* current_message;

if (f != NULL)
{

    while ( !feof(f) )
    {
        fgets(current_line, sizeof(current_line), f);

        if (strlen(current_line) > 4)
        {
            if (LineStartsWith(current_line, "BO_"))
            {
                current_message = AddMessage(current_line);
            }
            else if (LineStartsWith(current_line, " SG_"))
            {
                AddSignal(current_message, current_line);
            }
            else if (LineStartsWith(current_line, "CM_ SG_"))
            {
                AddSignalComment(current_line);
            }
            else if (LineStartsWith(current_line, "BA_ \"GenMsgCycleTime\" BO_"))
            {
                AddSendInterval(current_line);
            }
            else if (LineStartsWith(current_line, "BA_ \"SignalType\" SG_"))
            {
                AddSignalType(current_line);
            }
            else if (LineStartsWith(current_line, "BA_ \"SignalLongName\" SG_"))
            {
                AddSignalLongName(current_line);
            }
            else if (LineStartsWith(current_line, "VAL_"))
            {
                AddEnums(current_line);
            }
```

```
            }
        }

    }

    fclose(f);

}


MESSAGE* CanFormat::AddMessage(char* current_line)
{
    //BO_ Id MsgName: Bytes Transmitter


    MESSAGE* m = new MESSAGE;

    sscanf(current_line, "BO_ %d %s %d %s ",
        &(m->id), // id
        m->name, // name (remove : after)
        &(m->bytes),
        m->transmitter );

    m->name[strlen(m->name)-1] = '\0';

    messages.Add(m);

    return m;

}

void CanFormat::AddSignal(MESSAGE* m, char* current_line)
{
    // SG_ SigName : StartBit|BitLen@0+ (resolution,offset) [min|max] "units" Receiver

    SIGNAL* s = new SIGNAL;

    char sign;
    char trash;


    sscanf(current_line, " SG_ %s : %d|%d@0%c (%lf%c%lf) [%lf%c%lf] \"%[^\"]s ",
        s->name,
        &(s->bit_start),
```

```
        &(s->bit_length),
        &sign,
        &(s->resolution),
        &trash,
        &(s->offset),
        &(s->min),
        &trash,
        &(s->max),
        s->units//,
        //s->receivers
        );

    // get 'receivers'
    int sl = strlen(current_line);
    int i = sl - 2;
    while (i >= 0 && current_line[i] != ' ')
    {
        i--;
    }

    int j;
    for (j=(i+1); j<sl-1; j++)
    {
        s->receivers[j-i-1] = current_line[j];
    }
    s->receivers[j-i-1] = '\0';




    s->is_signed = (sign == '-');

    s->parent_data = m->data;

    m->signals.Add(s);

}

void CanFormat::AddSignalComment(char* current_line)
{
    // CM_ SG_ Id SigName "comment";

    int id;
```

```
    char signame[STRLEN_NAME];
    char* comment = new char[STRLEN_COMMENT];

    sscanf(current_line, "CM_ SG_ %d %s \"%[^\"]s ",
        &id,
        signame,
        comment);


    // find signal using id and name, then add the comment
    SIGNAL* s = FindSignal(id, signame);
    if (s != 0)
    {
        if (s->comment) delete[] s->comment;
        s->comment = comment;
    }

}

void CanFormat::AddSendInterval(char* current_line)
{
    //BA_ "GenMsgCycleTime" BO_ Id Interval

    int id;
    int interval;

    sscanf(current_line, "BA_ \"GenMsgCycleTime\" BO_ %d %d",
        &id,
        &interval );

    MESSAGE* m = FindMessage(id);
    if (m != 0) m->interval = interval;
}


void CanFormat::AddSignalType(char* current_line)
{
    // BA_ "SignalType" SG_ Id SigName "TYPE";

    int id;
    char signame[STRLEN_NAME];
    char* type = new char[STRLEN_TYPE];

    sscanf(current_line, "BA_ \"SignalType\" SG_ %d %s \"%[^\"]s",
```

```
        &id,
        signame,
        type);

    SIGNAL* s = FindSignal(id, signame);
    if (s != 0)
    {
        if (s->type) delete[] s->type;
        s->type = type;
    }
}


void CanFormat::AddSignalLongName(char* current_line)
{
    // BA_ "SignalLongName" SG_ Id SigName "LONG NAME";

    int id;
    char signame[STRLEN_NAME];
    char* longname = new char[STRLEN_LONGNAME];

    sscanf(current_line, "BA_ \"SignalLongName\" SG_ %d %s \"%[^\"]s",
        &id,
        signame,
        longname);

    SIGNAL* s = FindSignal(id, signame);
    if (s != 0)
    {
        if (s->long_name) delete[] s->long_name;
        s->long_name = longname;
    }
}


void CanFormat::AddEnums(char* current_line)
{
    // VAL_ Id SigName 0 "val1" 1 "val2" ;

    int id;
    char signame[STRLEN_NAME];
    char testsize[200];
    char format[100];
    int start;
```

```
    ENUM* e;

    sscanf(current_line, "VAL_ %d %s ",
        &id,
        signame );

    SIGNAL* s = FindSignal(id, signame);
    if (s != 0)
    {
        sprintf(testsize, "VAL_ %d %s ", id, signame);
        start = strlen(testsize);

        while (start < strlen(current_line) - 2)
        {
            e = new ENUM;
            e->name = new char[STRLEN_ENUM];

            sprintf(format, "%s%d%s", "%*", start, "c%d \"%[^\"]s");
            sscanf(current_line, format, &(e->value), e->name);

            s->enumerations.Add(e);

            sprintf(testsize, "%d \"%s\" ", e->value, e->name);
            start += strlen(testsize);
        }

    }

}

MESSAGE* CanFormat::FindMessage(int id)
{
    LINK<MESSAGE>* seeker = messages.head;
    while (seeker != NULL)
    {
        // do something on  *seeker->value
        if ( id == seeker->value->id )
        {
            return seeker->value;
        }

        seeker = seeker->next;
    }
```

```
    return 0;
}

MESSAGE* CanFormat::FindMessage(const char* name)
{
    LINK<MESSAGE>* seeker = messages.head;
    while (seeker != NULL)
    {
        // do something on  *seeker->value
        if ( 0 == strcmp(name, seeker->value->name) )
        {
            return seeker->value;
        }

        seeker = seeker->next;
    }

    return 0;
}

SIGNAL* CanFormat::FindSignal(int msg_id, const char* sig_name)
{
    MESSAGE* m = FindMessage(msg_id);
    if (m == 0)
    {
        return 0;
    }
    else
    {
        return m->FindSignal(sig_name);
    }
}

SIGNAL* CanFormat::FindSignal(const char* msg_name, const char* sig_name)
{
    MESSAGE* m = FindMessage(msg_name);
    if (m == 0)
    {
        return 0;
    }
    else
    {
        return m->FindSignal(sig_name);
    }
```

```
    return 0;
}

SIGNAL* CanFormat::FindSignal(const char* sig_name)
{
    LINK<MESSAGE>* m_finder;
    LINK<SIGNAL>* s_finder;

    m_finder = messages.head;
    while (m_finder != 0)
    {
        s_finder = m_finder->value->signals.head;
        while (s_finder != 0)
        {
            if (0 == strcmp(sig_name, s_finder->value->name))
            {
                return s_finder->value;
            }
            s_finder = s_finder->next;
        }

        m_finder = m_finder->next;
    }

    return 0;
}



bool CanFormat::LineStartsWith(char* line, const char test[])
{
    int testlen = strlen(test);

    if (strlen(line) < testlen)
    {
        return 0;
    }

    for (int i=0; i<testlen; i++)
    {
        if (line[i] != test[i])
        {
            return 0;
        }
```

```
    }

    return !0;
}

void CanFormat::print()
{
    LINK<MESSAGE>* seeker = messages.head;
    while (seeker != NULL)
    {
        seeker->value->print();
        seeker = seeker->next;
    }
}

bool CanFormat::Receive(int id, BYTE data[])
{
    MESSAGE* m = FindMessage(id);
    if (m != NULL)
    {
        return m->Receive(data);
    }
    return 0;
}

MESSAGE* CanFormat::ReadNext()
{
    // file should already be open
    int id;
    DWORD time;
    BYTE data[8];

    fread( &time, sizeof(id), 1, log_file );
    fread( &id, sizeof(time), 1, log_file );
    fread( data, 1, 8, log_file );

    MESSAGE* m = FindMessage(id);
    if (m != NULL)
    {
        m->receive_time = time;
        memcpy(m->data, data, 8);
    }

    return m;
```

```
}

void CanFormat::OpenLogFile(char* path)
{
    log_file = fopen(path, "rb");
}

void CanFormat::CloseLogFile()
{
    fclose(log_file);
}

bool CanFormat::LogFileFinished()
{
    return (feof(log_file) > 0);
}

void CanFormat::PrintLogFile(char* path)
{
    OpenLogFile(path);
    MESSAGE* m;
    while ( !LogFileFinished() )
    {
        m = ReadNext();

        if (m != NULL)
        {
            printf("%d [0x%x] %02X %02X %02X %02X %02X %02X %02X %02X\n",
                m->receive_time,
                m->id,
                m->data[0],
                m->data[1],
                m->data[2],
                m->data[3],
                m->data[4],
                m->data[5],
                m->data[6],
                m->data[7] );
        }

    }
    CloseLogFile();
}
```

```
MESSAGE* CanFormat::GetNextSendable()
{
    LINK<MESSAGE>* result = messages.head;
    while (result != 0)
    {
        if (result->value->send_enable)
        {
            if (result->value->send_time + result->value->interval < GetTickCount())
            {
                return result->value;
            }
        }
        result = result->next;
    }
    return 0;
}

void CanFormat::ResetData()
{
    for (LINK<MESSAGE>* m = messages.head; m != 0; m = m->next)
    {
        memset(m->value->data, 0, 8);
    }
}
```

A.3.7 Message.h

```
#pragma once

#include "Signal.h"
#include "windows.h"


#define STRLEN_TRANSMITTER 100


class MESSAGE
{
public:
    MESSAGE(void);
    ~MESSAGE(void);

    int id;
```

```
    char* name;
    int bytes;
    char* transmitter;
    int interval;
    bool send_enable;

    BYTE data[8];

    DWORD receive_time;
    DWORD send_time;

    LIST<SIGNAL> signals;

    SIGNAL* FindSignal(const char* name);

    bool Receive(BYTE new_data[]);
    void AppendToLogFile(char* path);

    void print();
};
```

A.3.8 Message.cpp

```
#include "StdAfx.h"
#include "Message.h"


MESSAGE::MESSAGE(void)
{
    id = 0;
    name = new char[STRLEN_NAME];
    strcpy(name, "");
    bytes = 0;
    transmitter = new char[STRLEN_TRANSMITTER];
    strcpy(transmitter, "");
    interval = 0;
    receive_time = 0;
    send_time = 0;
    send_enable = 0;

    //for (int i=0; i<sizeof(data); i++) data[i] = 0;
    memset(data, 0, 8);
```

```
}

MESSAGE::~MESSAGE(void)
{
    delete[] name;
    delete[] transmitter;
}


SIGNAL* MESSAGE::FindSignal(const char* name)
{

    LINK<SIGNAL>* seeker = signals.head;
    while (seeker != NULL)
    {
        // do something on  *seeker->value
        if ( 0 == strcmp(name, seeker->value->name) )
        {
            return seeker->value;
        }

        seeker = seeker->next;
    }

    return 0;
}

void MESSAGE::print()
{
    printf("=====================================\n");
    printf("%s (%d)\n", name, id);

    LINK<SIGNAL>* seeker = signals.head;
    while (seeker != NULL)
    {
        seeker->value->print();
        seeker = seeker->next;
    }
}

bool MESSAGE::Receive(BYTE new_data[])
{
    if (send_enable) return 0;
```

```
    receive_time = GetTickCount();

    if (0 == memcmp(new_data, data, 8)) return 0;

    memcpy(data, new_data, 8);

    return 1;
}

void MESSAGE::AppendToLogFile(char* path)
{
    // write receive time, id, and data bytes

    FILE* f = fopen( path, "ab" );

    fwrite( & receive_time, sizeof(receive_time), 1, f );
    fwrite( & id, sizeof(id), 1, f );
    fwrite( data, 1, 8, f );

    fclose( f );

}
```

A.3.9 Signal.h

```
#pragma once

#include <iostream>
#include "list.h"

//typedef char* STRING;


#define STRLEN_NAME          100
#define STRLEN_UNITS         20
#define STRLEN_RECEIVERS     100
#define STRLEN_COMMENT       200
#define STRLEN_TYPE          20
#define STRLEN_LONGNAME      200
#define STRLEN_ENUM          100


#define SIGDIR_TRANSMIT 1
```

```
#define SIGDIR_RECEIVE -1
#define SIGDIR_IGNORE 0

struct ENUM
{
    int value;
    char* name;
};



class SIGNAL
{
public:
    SIGNAL(void);
    ~SIGNAL(void);

    char* name;
    int bit_start;
    int bit_length;
    bool is_signed;
    double resolution;
    double offset;
    double min;
    double max;
    char* units;
    char* receivers;
    char* comment;
    char* type;
    char* long_name;
    LIST<ENUM> enumerations;
    BYTE* parent_data;
    char direction;

    void Set(double x);
    double Get();

    void print();

    int FindEnumeration(const char* search);
    void FindEnumeration(int search, char* result);

};
```

A.3.10 Signal.cpp

```
#include "StdAfx.h"
#include "Signal.h"
#include "MsgManip.h"


SIGNAL::SIGNAL(void)
{
    name = new char[STRLEN_NAME];
    strcpy(name, "");
    bit_start = 0;
    bit_length = 0;
    is_signed = true;
    resolution = 0;
    offset = 0;
    min = 0;
    max = 0;
    units = new char[STRLEN_UNITS];
    strcpy(units, "");
    receivers = new char[STRLEN_RECEIVERS];
    strcpy(receivers, "");
    comment = new char[STRLEN_COMMENT];
    strcpy(comment, "");
    type = new char[STRLEN_TYPE];
    strcpy(type, "");
    long_name = new char[STRLEN_LONGNAME];
    strcpy(long_name, "");
    direction = SIGDIR_IGNORE;
}

SIGNAL::~SIGNAL(void)
{
    delete[] name;
    delete[] units;
    delete[] receivers;
    delete[] comment;
    delete[] type;
    delete[] long_name;
}


void SIGNAL::print()
{
```

```
    printf("%s %s (%f to %f) %f %f %d %d ",
        type,
        name,
        min,
        max,
        resolution,
        offset,
        bit_start,
        bit_length);

    LINK<ENUM>* seeker = enumerations.head;
    while (seeker != NULL)
    {
        printf("%d=%s ", seeker->value->value, seeker->value->name);

        seeker = seeker->next;
    }

    printf("\n");
}


int SIGNAL::FindEnumeration(const char* search)
{
    LINK<ENUM>* seeker = enumerations.head;
    while (seeker != NULL)
    {
        if (0 == strcmp(seeker->value->name, search))
        {
            return seeker->value->value;
        }
        seeker = seeker->next;
    }

    return 0;
}

void SIGNAL::FindEnumeration(int search, char* result)
{
    LINK<ENUM>* seeker = enumerations.head;
    while (seeker != NULL)
    {
        if (search == seeker->value->value)
        {
```

```
            strcpy(result, seeker->value->name);
            return;
        }
        seeker = seeker->next;
    }

    strcpy(result, "");
}


void SIGNAL::Set(double x)
{
    int bit = bit_start % 8;
    int byte = (bit_start - bit) / 8;

    SaveDouble(parent_data, x, byte, bit, bit_length, is_signed, resolution, offset);
}

double SIGNAL::Get()
{
    int bit = bit_start % 8;
    int byte = (bit_start - bit) / 8;

    return LoadDouble(parent_data, byte, bit, bit_length, is_signed, resolution, offset);
}
```

A.3.11 MsgManip.h

```
#pragma once

#include "stdio.h"

typedef unsigned __int64 UI64;



UI64 MakeMask(int bit_length);
unsigned char Mask(int bit_length);
void CopyBits(unsigned char* data, int byte_start, int bit_start, int bit_length, UI64* result);
bool GetBit(unsigned char* data, int byte_start, int bit_start);
double LoadDouble(unsigned char* data, int byte_start, int bit_start, int bit_length,
    bool is_signed, double resolution, double offset);
void SaveDouble(unsigned char* data, double value, int byte_start, int bit_start, int bit_length,
```

```
    bool is_signed, double resolution, double offset);
void SetBits(unsigned char* data, UI64 value, int byte_start, int bit_start, int bit_length);
double pow2(int y);



void print_data(unsigned char* data);
void print_data_bin(unsigned char* data);
void print_bit(unsigned char data, int bit);
void print_ui64_bin(UI64 data);
```

A.3.12 MsgManip.cpp

```
#include "stdafx.h"
#include "MsgManip.h"




UI64 MakeMask(int bit_length)
{
    if (bit_length >= 64 || bit_length < 0)
    {
        return 0xFFFFFFFFFFFFFFFF;
    }
    else
    {
        UI64 r = 1;
        r = r << bit_length;
        return r - 1;
    }
}

unsigned char Mask(int bit_length)
{
    char result = 1;
    for (int i=1; i<bit_length; i++)
        result = result * 2 + 1;
    return result;
}


void CopyBits(unsigned char* data, int byte_start, int bit_start, int bit_length, UI64* result)
```

```
{
    //// copy to a UI64
    //unsigned char temp[8];
    //for (int i=0; i<8; i++) temp[i] = data[7-i];
    //memcpy( result, temp, 8 );

    //int shift_amount = 8 * byte_start + bit_start - bit_length + 1;

    //*result = (*result >> shift_amount) & MakeMask(bit_length);

    if (bit_length-1 <= bit_start)
    {
        *result = ( data[byte_start] >> (bit_start-bit_length+1) ) & Mask(bit_length);
    }
    else
    {
        *result = data[byte_start] & Mask(bit_start + 1);
        bit_length -= (bit_start + 1);
        byte_start--;

        while (bit_length > 8)
        {
            *result = (*result) << 8;
            *result |= data[byte_start];
            bit_length -= 8;
            byte_start--;
        }

        *result = (*result) << bit_length;
        *result |= (data[byte_start] >> (8-bit_length));
    }
}

bool GetBit(unsigned char* data, int byte_start, int bit_start)
{
    return ( ( data[byte_start] >> bit_start ) & 1 ) > 0;
}

double LoadDouble(unsigned char* data, int byte_start, int bit_start, int bit_length,
    bool is_signed, double resolution, double offset)
{
    // grab the bits
    UI64 bits = 0;
    CopyBits(data, byte_start, bit_start, bit_length, &bits);
```

```
    // if is_signed and sign bit set
    if (is_signed && GetBit(data, byte_start, bit_start))
    {
        // convert to 2's compliment
        bits = ( (~bits) + 1 ) & MakeMask(bit_length);
        return (resolution * bits) * -1 + offset;
    }
    else
    {
        return bits * resolution + offset;
    }

}

void SaveDouble(unsigned char* data, double value, int byte_start, int bit_start, int bit_length,
    bool is_signed, double resolution, double offset)
{
    double min, max;

    double range = (pow2(bit_length) - 1);
    range *= resolution;
    if (is_signed)
        min = -1.0 * pow2(bit_length-1) * resolution;
    else
        min = offset;
    max = min + range;


    if (value < min) value = min;
    if (value > max) value = max;


    UI64 bits = 0;
    if (is_signed && (value-offset) < 0)
    {
        value = value - offset;
        value = -1 * value;
        value = value / resolution;
        value = value + 0.5;

        bits = value;
```

```
        //bits = (double) (((value - offset) / (-1 * resolution)) + 0.5);
        bits = (~bits) + 1;
    }
    else
    {
        value = value - offset;
        value = value / resolution;
        value = value + 0.5;

        bits = value;
    }

    bits = bits & MakeMask(bit_length);


    SetBits(data, bits, byte_start, bit_start, bit_length);
}

void SetBits(unsigned char* data, UI64 value, int byte_start, int bit_start, int bit_length)
{

    if (bit_length <= bit_start)
    {
        // make a hole
        data[byte_start] &= ~(Mask(bit_length) << (bit_start - bit_length + 1));
        // add the data
        data[byte_start] |= value << (bit_start - bit_length + 1);
    }
    else
    {
        UI64 temp = 0;

        // MSByte
        temp = (value >> (bit_length - bit_start - 1));
        data[byte_start] &= ~Mask(bit_start + 1);
        data[byte_start] |= temp;
        bit_length -= (bit_start + 1);
        byte_start--;

        while (bit_length > 8)
        {
            temp = (value >> (bit_length - 8));
            data[byte_start] = (temp & 0xFF);
```

```
            bit_length -= 8;
            byte_start--;
        }

        // LSByte
        temp = (value & Mask(bit_length)) << (8-bit_length);
        data[byte_start] &= ~(Mask(bit_length) << (8-bit_length));
        data[byte_start] |= temp;
    }

}

double pow2(int y)
{
    double result = 1.0;
    for (int i=0; i<y; i++)
    {
        result *= 2.0;
    }
    return result;
}









void print_data(unsigned char* data)
{
    printf("0x%02X%02X%02X%02X%02X%02X%02X%02X \n", data[7], data[6], data[5], data[4], data[3],
        data[2], data[1], data[0]);
}

void print_data_bin(unsigned char* data)
{
    printf("0b ");
    for (int i=7; i>=0; i--)
```

```
    {
        for (int j=7; j>=0; j--)
        {
            print_bit(data[i], j);
        }
        printf(" ");
    }
    printf("\n");
}

void print_bit(unsigned char data, int bit)
{
    if (data & (1 << bit))
        printf("1");
    else
        printf("0");
}

void print_ui64_bin(UI64 data)
{
    printf("0b ");
    UI64 mask = 1;
    mask = mask << 63;
    for (int i=63; i>=0; i--)
    {
        if (data & (mask))
            printf("1");
        else
            printf("0");

        if (i % 8 == 0)
            printf(" ");

        mask = mask >> 1;
    }
    printf("\n");
}
```

A.3.13 frmSimulate.h

```
#pragma once

#include "globals.h"
```

```
#include "FsmSimulation.h"
#include "xlCAN_All.h"

using namespace System;
using namespace System::ComponentModel;
using namespace System::Collections;
using namespace System::Windows::Forms;
using namespace System::Data;
using namespace System::Drawing;


namespace thesis_can {

    /// <summary>
    /// Summary for frmSimulate
    ///
    /// WARNING: If you change the name of this class, you will need to change the
    ///          'Resource File Name' property for the managed resource compiler tool
    ///          associated with all .resx files this class depends on.  Otherwise,
    ///          the designers will not be able to interact properly with localized
    ///          resources associated with this form.
    /// </summary>
    public ref class frmSimulate : public System::Windows::Forms::Form
    {
    public:
        frmSimulate(void)
        {
            InitializeComponent();
            //
            //TODO: Add the constructor code here
            //
        }

    protected:
        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        ~frmSimulate()
        {
            if (components)
            {
                delete components;
            }
        }
```

```
    private: System::ComponentModel::IContainer^  components;
    protected:
    private: System::Windows::Forms::Label^  label1;
    private: System::Windows::Forms::Label^  label2;
    private: System::Windows::Forms::Timer^  timer1;

    private:
        /// <summary>
        /// Required designer variable.
        /// </summary>


    private: FiniteStateMachine* fsm;
    private: FsmSimulation* simulator;
    private: System::Windows::Forms::PictureBox^  pictureBox1;
    private: System::Windows::Forms::TableLayoutPanel^  tableLayoutPanel1;


    private: long* hcan;

#pragma region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        void InitializeComponent(void)
        {
            this->components = (gcnew System::ComponentModel::Container());
            this->label1 = (gcnew System::Windows::Forms::Label());
            this->label2 = (gcnew System::Windows::Forms::Label());
            this->timer1 = (gcnew System::Windows::Forms::Timer(this->components));
            this->pictureBox1 = (gcnew System::Windows::Forms::PictureBox());
            this->tableLayoutPanel1 = (gcnew System::Windows::Forms::TableLayoutPanel());
            (cli::safe_cast<System::ComponentModel::ISupportInitialize^  >(
                this->pictureBox1))->BeginInit();
            this->tableLayoutPanel1->SuspendLayout();
            this->SuspendLayout();
            //
            // label1
            //
            this->label1->AutoSize = true;
            this->label1->Location = System::Drawing::Point(3, 0);
            this->label1->Name = L"label1";
            this->label1->Size = System::Drawing::Size(35, 13);
```

```
this->label1->TabIndex = 0;
this->label1->Text = L"label1";
//
// label2
//
this->label2->AutoSize = true;
this->label2->Location = System::Drawing::Point(53, 0);
this->label2->Name = L"label2";
this->label2->Size = System::Drawing::Size(35, 13);
this->label2->TabIndex = 1;
this->label2->Text = L"label2";
//
// timer1
//
this->timer1->Interval = 1;
this->timer1->Tick += gcnew System::EventHandler(this, &frmSimulate
    ::timer1_Tick);
//
// pictureBox1
//
this->tableLayoutPanel1->SetColumnSpan(this->pictureBox1, 2);
this->pictureBox1->Dock = System::Windows::Forms::DockStyle::Fill;
this->pictureBox1->Location = System::Drawing::Point(3, 23);
this->pictureBox1->Name = L"pictureBox1";
this->pictureBox1->Size = System::Drawing::Size(384, 261);
this->pictureBox1->TabIndex = 2;
this->pictureBox1->TabStop = false;
//
// tableLayoutPanel1
//
this->tableLayoutPanel1->ColumnCount = 2;
this->tableLayoutPanel1->ColumnStyles->Add((gcnew System::Windows::Forms
    ::ColumnStyle(System::Windows::Forms::SizeType::Absolute,
    50)));
this->tableLayoutPanel1->ColumnStyles->Add((gcnew System::Windows::Forms
    ::ColumnStyle(System::Windows::Forms::SizeType::Percent,
    100)));
this->tableLayoutPanel1->Controls->Add(this->label1, 0, 0);
this->tableLayoutPanel1->Controls->Add(this->label2, 1, 0);
this->tableLayoutPanel1->Controls->Add(this->pictureBox1, 0, 1);
this->tableLayoutPanel1->Dock = System::Windows::Forms::DockStyle::Fill;
this->tableLayoutPanel1->Location = System::Drawing::Point(0, 0);
this->tableLayoutPanel1->Name = L"tableLayoutPanel1";
this->tableLayoutPanel1->RowCount = 2;
```

```
            this->tableLayoutPanel1->RowStyles->Add((gcnew System::Windows::Forms
                ::RowStyle(System::Windows::Forms::SizeType::Absolute, 20)));
            this->tableLayoutPanel1->RowStyles->Add((gcnew System::Windows::Forms
                ::RowStyle(System::Windows::Forms::SizeType::Percent, 100)));
            this->tableLayoutPanel1->Size = System::Drawing::Size(390, 287);
            this->tableLayoutPanel1->TabIndex = 3;
            //
            // frmSimulate
            //
            this->AutoScaleDimensions = System::Drawing::SizeF(6, 13);
            this->AutoScaleMode = System::Windows::Forms::AutoScaleMode::Font;
            this->ClientSize = System::Drawing::Size(390, 287);
            this->Controls->Add(this->tableLayoutPanel1);
            this->Name = L"frmSimulate";
            this->Text = L"Simulate";
            this->Load += gcnew System::EventHandler(this, &frmSimulate
                ::frmSimulate_Load);
            this->FormClosed += gcnew System::Windows::Forms
                ::FormClosedEventHandler(this, &frmSimulate::frmSimulate_FormClosed);
            (cli::safe_cast<System::ComponentModel::ISupportInitialize^  >(
                this->pictureBox1))->EndInit();
            this->tableLayoutPanel1->ResumeLayout(false);
            this->tableLayoutPanel1->PerformLayout();
            this->ResumeLayout(false);

        }
#pragma endregion
private: System::Void frmSimulate_Load(System::Object^  sender, System::EventArgs^  e)
{
    fsm = new FiniteStateMachine;
    simulator = new FsmSimulation;
    hcan = new long;

    Windows::Forms::OpenFileDialog^ ofd = gcnew Windows::Forms::OpenFileDialog;
    ofd->Title = "Open FSM File";
    ofd->FileName = "";
    ofd->Filter = "FSM Files (*.fsm)|*.fsm|All Files (*.*)|*.*";
    if (ofd->ShowDialog() != Windows::Forms::DialogResult::OK)
    {
        this->Close();
    }
    else
    {
        char* fsm_file_path = (char*)(void*)Runtime::InteropServices::Marshal
```

```
            ::StringToHGlobalAnsi(ofd->FileName);
        fsm->ReadXml(fsm_file_path);
        Runtime::InteropServices::Marshal::FreeHGlobal(IntPtr(fsm_file_path));
    }

    ofd->Title = "Open Picture File";
    ofd->FileName = "";
    ofd->Filter = "PNG Image (*.png)|*.png|All Files (*.*)|*.*";
    if (ofd->ShowDialog() == Windows::Forms::DialogResult::OK)
    {
        char* img_file_path = (char*)(void*)Runtime::InteropServices::Marshal
            ::StringToHGlobalAnsi(ofd->FileName);
        pictureBox1->ImageLocation = gcnew String(img_file_path);
        Runtime::InteropServices::Marshal::FreeHGlobal(IntPtr(img_file_path));
    }

    Windows::Forms::MessageBox::Show("Press OK to start simulation.");

    initialize_driver(CAN_VIRTUAL, hcan);

    simulator->dbc = &can_dbc;
    simulator->fsm = fsm;
    simulator->AdjustDbc();

    timer1->Enabled = true;


}
private: System::Void frmSimulate_FormClosed(System::Object^  sender, System::Windows
             ::Forms::FormClosedEventArgs^  e)
{
    close_driver(*hcan);
}

private: System::Void timer1_Tick(System::Object^  sender, System::EventArgs^  e)
{
    int status;
    int rxId;
    unsigned char rxData[8];
    char description[100];

    if (!simulator->Done())
    {
        label1->Text = String::Format("S{0}", simulator->GetCurrentState());
```

```
        simulator->GetMode(description);
        label2->Text = gcnew String(description);
        simulator->Service();
        status = 0;

        for (int i=0; (i<10) && (status == 0); i++)
        {
            status = receive_data(rxData, &rxId, *hcan);
            if (0 == status)
            {
                can_dbc.Receive(rxId, rxData);
            }
        }

        sendAllCan();
    }
    else
    {
        timer1->Enabled = false;
    }
}

private: void sendAllCan()
{
    for (LINK<MESSAGE>* m = can_dbc.messages.head; m != 0; m = m->next)
    {
        if ( (m->value->send_enable) && (GetTickCount() > (m->value->interval +
            m->value->send_time)) )
        {

            transmit_data(m->value->data, m->value->id, *hcan);
            m->value->send_time = GetTickCount();
        }
    }
}

}; // end form
}
```